1984

The enhancement of concurrent processing through functional programming languages.

McGrath, Thomas R.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE ENHANCEMENT OF CONCURRENT PROCESSING
THROUGH
FUNCTIONAL PROGRAMMING LANGUAGES

by

Thomas R. McGrath

June 1984

Thesis Advisor: Bruce J. MacLennan

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
The Enhancement of Concurrent Processing through Functional Programming Languages

5. TYPE OF REPORT & PERIOD COVERED
Master's Thesis
June 1984

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Thomas R. McGrath

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Postgraduate School
Monterey, CA 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Postgraduate School
Monterey, CA 93943

12. REPORT DATE
June 1984

13. NUMBER OF PAGES
74

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Parallel Processing, Functional, Concurrent Processing, Multiprocessor

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
The "von Neumann bottleneck" imposes severe limitations on programming languages. This thesis points out that although the hardware limitations imposed by this bottleneck are being overcome, its constraints will remain in programs as long as there are assignment statements in their code. We assert that functional programming languages allow us to harness the processing power of computers with hundreds or even thousands of

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

processors, and also allow us to solve problems which are time/cost prohibitive on a uniprocessor.
We discuss a mechanical method for transforming imperative programs into functional programs. We feel that the mechanical transformation process is very inexpensive, and that it might be the best way to make imperative "library" programs into functional ones which are well suited to concurrent processing.

Approved for public release; distribution unlimited.

The Enhancement of Concurrent Processing through Functional Programming Languages

by

Thomas R. McGrath
Lieutenant Commander, United States Navy
B.S., Cornell University, 1968
M.S.S.M, University of Southern California, 1981

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1984

ABSTRACT

The "von Neumann bottleneck" imposes severe limitations on programming languages. This thesis points out that although the hardware limitations imposed by this bottleneck are being overcome, its constraints will remain in programs as long as there are assignment statements in their code. We assert that functional programming languages allow us to harness the processing power of computers with hundreds or even thousands of processors, and also allow us to solve problems which are time/cost prohibitive on a uniprocessor.

We discuss a mechanical method for transforming imperative programs into functional programs. We feel that the mechanical transformation process is very inexpensive, and that it might be the best way to make imperative "library" programs into functional ones which are well suited to concurrent processing.

TABLE OF CONTENTS

LIST OF FIGURES

I. HISTORY AND INTRODUCTION

The von Neumann architecture was a brilliant breakthrough in the development of computers. Through this design computing machines achieved an execution speed and power which was foreseen only by men of great vision. Unfortunately, word-at-a-time processing, which is implicit in this architecture, has become a limiting factor in the advancement of machine speed.

The so-called von Neumann bottleneck can be overcome in computer architecture. Indeed, there are many architectures which employ a variety of techniques to circumvent the bottleneck, by using multiple buses along with multiple central processing units (CPUs). For many decades, commercial computers have been structured to handle information sequentially. Now, scientists are trying to replace the large computer, based on serial instructions, with networks of small computers linked in a way that would enable them to work on different parts of a problem concurrently [Ref. 1]. Many experts forecast that Japan's fifth generation computer systems will make the Smithsonian Institute the only appropriate place in which to house von Neumann machines. These new computers virtually eliminate von Neumann bottlenecks [Ref. 2].

Not so obvious is the fact that the von Neumann bottleneck has become manifest not only in computer architecture, but in the languages which were designed with these machines in mind. Since the development of Fortran in the early 1950's, high-level programming languages have been based in large part on the instruction sets of their "target machines". Fortran was a very efficient language, and it achieved that efficiency because its optimizer was developed

with the instruction set of the IBM 701 in the forefront of the designer's mind [Ref. 3: p.33]. Since that time, the von Neumann bottleneck has firmly established itself in every imperative programming language. The bottleneck is manifested in the form of assignment statements [Ref. 4]. We thus find ourselves in a situation where the high-level languages we ordinarily use are not capable of taking advantage of the computing power of state-of-the-art machines. It seems obvious that computing power which cannot be harnessed is not of much value to us. What can we do about that? This is the question which provided the motivation for this thesis.

Since high-level languages have been to a very great extent designed with the instruction sets of their target machines in mind, there are limitations built into the structure of the languages which will be very difficult to overcome. I will spend some time focusing on the weaknesses of the imperative languages. I would like to say at the outset, however, that I in no way mean to imply that imperative languages are not extremely useful in many applications. The limitations on which I will primarily focus will be in terms of imperative languages as applied to concurrent processing.

Similarly, I will discuss functional programming languages. I believe that they provide some relief from the von Neumann bottleneck that cannot be achieved by working within the framework of imperative languages. Put another way, I believe that functional programming languages will enable the user to take advantage of the power afforded by these new multiprocessor machines in a way that imperative languages simply cannot. Indeed there are techniques which can be employed which will extend the "concurrent processing power" of imperative languages, but these techniques will never manifest all the advantages that functional languages afford, such as evaluation order independence.

There are techniques available which allow for imperative programs to be translated into functional ones [Ref. 5: pp. 136-149]. I will demonstrate one of these techniques on a widely used imperative program: the Shell sort. It should be noted, however, that functional programs also have their limitations; but these limitations seem to apply to areas other than the concurrent processing issue.

Aside from the fact that multiprocessor hardware is becoming available, there is another important reason for wanting to develop and exploit the properties of functional programming languages which enhance concurrent processing. Research conducted for NASA by an independent research firm [Ref. 6] discusses a whole class of problems that are today too computationally complex to be accomplished using conventional computer resources. For example the linear static analysis of an undersea oil platform was conducted using finite-element structural analysis. The problem had over 720,000 degrees of freedom, and took about one week of processing time on a Univac 1110 computer [Ref. 6: pp. 7-8]. The same authors point out that in the data-flow machine operators "fire" as soon as their operands are available. This is exactly how functional programs work: a function "fires" as soon as all of its parameters are available! Although programs expressed in sequential languages have been successful at expressing parallelism to some degree, they do not appear to have the potential of detecting parallelism of a high degree (100 or more processors) [Ref. 6: p.20].

As seems so often to be the case in computer science issues, no one technique will serve as a panacea. Functional programming is no exception. Rather, it provides the user with a great number of advantages, particularly in the area of concurrent processing. These must be weighed against the disadvantages, and a decision can then be made based on the

specific application for which the program is intended. I hope that this paper will help provide the background for that decision-making process.

II. IMPERATIVE LANGUAGES: STRENGTHS AND LIMITATIONS

A. CONVERSATIONS WITH MACHINES

Computers , if properly directed, have the ability to execute a great many instructions in a relatively short period of time. Yet in order to harness this computational power, one must be able to communicate with the computer, and give it some "marching orders". For quite some time, the only way in which to effectively communicate with computing machines was to use the machine's native language (cleverly dubbed "machine language"). Indeed, many people learned to use machine language very well, and some even began to *like* it!!! To most people, however, talking to a machine was quite a *strange* concept. Talking using an alphabet consisting of only zeros and ones was even more bizarre! There seemed to be two camps which developed from this "language problem". One camp *lived* for computers. They were convinced that the future of the world belonged to those who could "speak" machine language, and spared no effort in becoming friends with their inanimate associates. The other camp was at the same time enamored with, skeptical of, and intimidated by these new machines. These people swore that the slide rule would never be replaced, and that the computers were more trouble than they were worth.

To some extent, both camps were right. Computers certainly do have the the ability to complete tedious, boring tasks very quickly and very accurately. Even today, however, it seems that it can sometimes be more trouble than it is worth to get the machine to do what we want. In fact, sometimes it seems that we are working for the computer, instead of the computer serving us as it should be. The

development of high-level languages was undertaken in large part to narrow the gap between the two camps described above.

B. ADVANTAGES OF HIGH-LEVEL LANGUAGES

The fundamental purpose of high-level languages is to provide people with a more natural way to communicate with machines. High-level languages enable people to raise their communications to a higher level of abstraction, and to rely on an interpreter or compiler to translate their program into machine language. When developing a high-level language, it is important to ask the question, "For whom should the programming language be designed, anyway?" Of course, the answer is that it should serve its (human) user. As obvious as that seems, there are still a great many instances when that principle is not at the forefront of the designer's mind, and the user ends up "working" for the machine to some extent. C.A.R. Hoare has never stopped preaching the need to keep the human user in mind when dealing in programming language design [Ref. 7] [Ref. 8]. High-level languages should be kept as simple as possible. Each extra "feature" added to a language is one more thing that the user has to learn. In order to justify the inclusion of a feature in a language, the contribution that it makes should overwhelmingly outweigh the complexity it adds to the language.

High-level languages bridge the gap between natural (human) languages and machine languages. In the best case, therefore, programming languages should be the same as natural languages. According to Winograd [Ref. 9] the ultimate programming language would be one in which the programmer writes only the comments, and the programming environment would take it from there. In other words, the

user would be able to use a natural (spoken) language, and the system would take care of converting that to the language of the target machine.

Although this goal seems unachieveable, it is certainly something for which we should strive. We should make every effort to make programming languages understandable (to the human), and at the same time keep error-checking features, such as strong typing, embedded in them.

High-level programming languages free the user from some of the details of machine implementation, and hence these languages are more powerful and understandable than machine languages. Figure 2.1 illustrates a simple "add" instruction written in four ways: machine language [Ref. 10], assembly language [Ref. 10], high-level language, and natural language.

Another advantage of high-level languages is that they are transportable, i.e., they can be used on more than one type (brand/model) of machine. Compilers and interpreters take care of translating them into the instruction set of the target machine. Programs written in high-level languages are therefore easier to maintain throughout their life cycle.

Through the years, high-level languages have become more powerful and more understandable. In the next section I will discuss the evolution of high-level languages.

C. THE EVOLUTION OF IMPERATIVE LANGUAGES

With Figure 2.1 in mind,[1] it's hard to imagine how people put up with machine language for so long! As we shall see, successive generations of high-level languages have

---

[1]Note that Figure 2.1 is an extremely simple example. When conditional expressions, looping, and recursion are introduced, the differences in complexity among the different types of languages become even more pronounced.

Machine Language (Intel 8080)

```
10110111
00000110
00011001
00111110
00000111
10000000
11010011
11011000
01110110
```

Assembly Language (Intel 8080)

```
SUB A       ; clear accumulator
MVI B, 25D  ; place 25 in B register
MVI A, 7D   ; place 7 in accumulator
ADD B       ; sum of 7 and 25 placed in accumulator
OUT D8      ; print 32 (D8 is port to printer)
HLT         ; stop program
```

High Level Language

```
Begin
 A := 25;
 B := 7;
 Sum := A + B;
 WRITELN (Sum)
End.
```

Natural Language

Print the sum of 25 and 7.

Figure 2.1 Language Comparisons for a Simple "Add"

made programming much easier, but many feel it is still too complex and tedious for the average user to pick up. Thus the ultimate users of computing power--businessmen, accountants, scientists and engineers--still require a middleman to communicate with their machines [Ref. 12].

As we quickly look at the development of imperative programming languages, let's keep in mind the attributes

these languages should have,[2] some of which are listed in Figure 2.2.

- easy to learn
- easy to understand
- transportable from machine to machine
- free the user from mundane tasks
- enable the user to work at a higher level of abstraction
- do what the user intends

Figure 2.2 Desired Attributes of High Level Languages

People in all walks of life seem to resist change. Those computer scientists who were "comfortable" with machine language embraced the concept of the assembler, since it made coding easier, and translated directly into machine language. This helped the transportability of the program, since a given program could be run on a different machine once it was reassembled. The concept of high-level languages, however, was not so readily accepted by these scientists.

The principal objection to high-level languages was that they degraded machine efficiency, and hence a significant portion of the speed advantage of the computer would be needlessly and wastefully lost. FORTRAN was able to gain acceptance because it generated code that could usually equal, and sometimes surpass the efficiency of code generated by hand [Ref. 3: pp.33-34]. FORTRAN employed sophisticated optimization techniques. That, coupled with the fact

---

[2]For a more complete discussion on the development of attributes in programming languages, see MacLennan's work [Ref. 3]. I am not considering such things as Parnas' principle of information hiding, but rather will focus on the understandability of the language and the degree to which it lends itself to concurrent processing.

that it was designed specifically to be implemented on the IBM 704, allowed it to achieve an efficiency greater than many current-day programming languages. It is extremely important to note that the design of the programming language followed the design of the machine. This is a trend which has remained throughout the evolution of imperative programming languages. It was quite a reasonable dependency at the time that FORTRAN was developed, since computer hardware was much more costly than computer software. This trend has been reversed [Ref. 28], however, so at least from the viewpoint of cost, we are now free to develop languages without specific hardware configurations in mind...and then develop the hardware based on the software requirements.

FORTRAN had a tremendous impact on the computer science industry. It certainly freed the user from many mundane tasks, and enabled him/her to work at a higher level of abstraction. However after the "honeymoon" of FORTRAN was over; ways in which it could be improved began to surface. In 1968, Dijkstra stated that he was convinced that the go to should abolished from high-level languages [Ref. 14]. He felt that the go to statement was an invitation to make a mess of one's program, since it was so unstructured. ALGOL-60 had many features which potentially made programs much easier to understand, and hence easier to maintain. Indeed, Wulf [Ref. 15] developed a systematic way to eliminate go tos from a program, by introducing Boolean variables. Wulf was among many who seemed to feel that efficiency should not be maximized at the expense of understandability of a program. There seemed to be a strong (and in my view healthy) trend toward developing languages that were "user friendly." This trend continued with the design of Pascal, which was developed as a "teaching language". It also encorporated strong typing and parameter passing safeguards in order to protect the user from program side effects.

Shortly after Pascal was developed, Wulf and Shaw declared that global variables were also harmful [Ref. 16]. He pointed out that they also lead to side effects, and were really a result of sloppy (and lazy) programming.

Throughout the development of imperative programming languages, a strong effort was made to have them serve their (human) users by making them easier to learn and to understand. At the same time, very large scale integration (VLSI) circuits were being developed, which was making computer hardware both more efficient and less expensive. This was part of the reason why machine efficiency could be sacrificed for the sake of language clarity. John Backus (ironically, the man behind the design of FORTRAN) pointed out in 1978 [Ref. 4] that imperative languages were slaves to the word-at-a-time architecture on which they were originally developed. He tagged one more construct as being harmful: the assignment statement!

D. THE VON NEUMANN BOTTLENECK OF PROGRAMMING LANGUAGES

Although the von Neumann architecture was a brilliant breakthrough in the development of computer systems, its function relies on the transfer of information between memory and the central processing unit along a bus. Inherent in this architecture is the fact that information flow is limited to one word at a time. Unfortunately, this limitation (known as the von Neumann bottleneck) has put an upper bound on the potential speed of conventional computers.

Remember that one of the reasons that FORTRAN was so efficient was that the designers of its optimizer used the instruction set of the target machine as the frame of reference from which they worked. As successive generations of languages were developed, designers depended less and less on the architecture of the target machine(s). However, in

most cases, languages were still developed using the von Neumann architecture as the general developmental framework. Backus points out [Ref. 4] "...programming languages use variables to imitate the computer's storage cells. Control statements elaborate its jump and test instructions, and assignment statements imitate its fetching, storing, and arithmetic. The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does."

Backus goes on to describe how imperative languages have stifled the creativity of computer architects, since many architects are in a way held prisoner by the von Neumann mindset. Moreover, even languages which have attempted to avoid the imperative features (such as LISP) have been engulfed in von Neumann features.[3] It would seem that there is a vicious circle between the architecture bottleneck and the language bottleneck. If so, then why aren't imperative languages good enough?

The reason is that many computer architects have abandoned the von Neumann concepts in their designs, and are coming up with designs which can potentially process information much faster than conventional machines. Lerner points out that the advent of VLSI technology has made the development of highly parallel computers a practical possibility [Ref. 17]. He says, "Of the various competing ideas of how a parallel computer can be built, the best known and most developed is called data-flow. In data-flow computers, each of many identical processors calculates results as the data for a given computation becomes available."

---

[3]LISP has features such as "PROG" and "SETQ" which are really forms of the assignment statement. In Chapter IV I give an example of a LISP program which illustrates this point.

The groundwork has been laid for concurrent processing. In order to utilize the potential of parallel processors, the bottleneck of programming languages must be eliminated, or at least reduced. This has been a topic of considerable discussion,[4] particularly in operating systems, where the concept of "processes" is used. There are at least three difficulties encountered with the concurrent process concept: communication, synchronization, and non-determinancy. There is an excellent discussion of these by Bryant and Dennis, using the airline scheduling problem as an example [Ref. 19]. Dijkstra describes a system of "cooperating" sequential processes in which he uses two semaphores, "P operation" and "V operation" to permit concurrency and eliminate side effects [Ref. 18]. There are difficulties posed by this system: the processes must be cooperating, and there exists danger of a "deadly embrace" (deadlock).

Hoare describes a system of monitors which assumes (as in the case of semaphores) that all processes have access to a single shared memory [Ref. 20]. Both semaphores and monitors provide a means to suspend the execution of processes until certain conditions are satisfied. Problems of deadlock remain an issue. Actor semantics is another way of enhancing parallel processing through message passing [Ref. 21].

All of these methods are attempts to extend the power of imperative languages. They try to circumvent the limitations of the assignment statement, rather than dealing with it directly. In order to fully utilize the computational/processing power of parallel machines, parallelism must be built into the languages themselves.

---

[4]Concurrent processing is not a new concept, but it becomes even more important in view of the breakthroughs that are being made in computer architecture.

An extension to Pascal was established with just this purpose in mind. Essentially, semaphores were made available for Pascal, which allowed the programmer to take advantage of concurrency. Pascal has no built-in support for concurrency. It is the responsibility of the programmer to identify critical sections[5] and to "protect them" with P and V operations.

The difficulty with utilizing the above method to write concurrent program sections is that it forces the programmer to think at too detailed a level. That makes the chance of creating an error (and perhaps one which will be manifested only in subtle but important side effects) all too great.

There are two main issues in programming languages which support concurrent processing [Ref. 19]. The first is that the expressive power of the language should be maximized. The second is that programs should be clear and understandable. The latter is especially important in concurrent programming languages.

Ada is another example of an imperative language which attempts to make concurrency more attainable. Its designers seem to recognize that the assignment statement is directly related to the concurrent processing limitations of imperative languages. Booch suggests that therein lies the strength of Ada: a program designer can take a declarative view of the solution, not the imperative one that many other languages force them into [Ref. 22]. The basic construct for concurrent processes in Ada is the task. A task is like a package, but instead of types, constants, variables, procedures, and functions, a task exports only task entries. Task entries correspond most closely to procedures with in,

---

[5]A critical section is a piece of program belonging to a class of program sections of which only one can be executed at a time. In other words critical sections are interdependent. In order for program sections to run concurrently, they need to have mutually exclusive access to critical sections they reference [Ref. 25].

out, or in-out parameters. The implementation of a task is hidden from the user in the same manner as a package body. Task bodies describe the necessary synchronization of the implemented entries [Ref. 23].

The task concept does enhance concurrent processing at the course-grained level. Ada also encourages modularization, which from a design point of view, encourages the development of components which lend themselves to concurrent processing, i.e. are independent of one another. Also, the task is a built in feature of the language which directly supports concurrent processing.

In my view, however, Ada does not go far enough. When Dijkstra and others identified the go to as harmful [Ref. 14], the solution was not to reduce the number of go tos, but to eliminate them through structured programming. Similarly, programmers could be forced to a higher level of abstraction through a functional programming language (FPL) which eliminates the use of assignment statements [Ref. 24].

In my opinion, the best way to eliminate assignment statements and to maximize concurrent processing is through evaluation order independence. Functional programming languages exhibit this property. In the next chapter I will discuss evaluation order independence and functional programming languages in more detail.

III. FUNCTIONAL PROGRAMMING LANGUAGES: STRENGTHS

A. AN OVERVIEW OF FUNCTIONAL PROGRAMMING LANGUAGES

The functional programming to which I have been referring is known by a variety of names, such as applicative programming and value-oriented programming. It is a method of programming which differs from imperative programming in several important ways. As I have mentioned in previous chapters, imperative languages depend heavily on the assignment statement for accomplishing their tasks. MacLennan points out that most imperative programming languages are basically collections of mechanisms for routing control from one assignment statement to another. In a functional application, the central idea is to apply a function to its arguments [Ref. 3: pp.344-345]. This can be done in a variety of notations (discussed later) but is commonly expressed in Cambridge Polish. Cambridge Polish is also called prefix notation because the operator is written before the operands.[6] Functional notation quite naturally allows the programmer to raise himself to higher levels of abstraction. This is because functions can be applied to functions. (See Figure [3.8] for an example.) Functional programs also use "layering" to free the programmer from details. For example, in order to update an array, the programmer would simply call the function update [Ref. 24] (Figure 3.1). Such a function replaces the ith element of array A with x. The programmer need not concern himself with cons, rest, or the recursive nature of the function. He is able to concentrate on building programs rather than concentrating on the

---

[6] As a simple example, the infix algebraic expression (a+b) would be written as plus(a,b) in a prefix functional notation.

```
update(A,i,x) =
                if i=1 ---> cons(x, rest A)
                else cons[first A,
                          update(rest A, i-1, x)]
```

Figure 3.1 Functions Applied to Functions

objects which make up the program [Ref. 11]. This leads to programs which are more understandable, and hence easier to maintain. There is a cost involved though: program efficiency. Henderson estimates that functional programs may be as much as ten times less efficient than machine language[7] [Ref. 5].

To thoroughly discuss the development of a functional programming language is beyond the scope of this paper. Rather I will give a few simple examples. In the next chapter I will give examples of functional programs which are a bit more complicated. A more detailed explanation of the semantics of functional programming can be found in textbooks by Henderson [Ref. 5] or Burge [Ref. 26], or in MacLennan's soon-to-be published text [Ref. 24]. As I mentioned earlier, LISP has many functional features. Therefore, an understanding of functional programming semantics could also be achieved by studying LISP, although one would have to be careful to "filter out" the imperative features that it contains.

---

[7]This efficiency loss is due not only to compiler use, but also to the fact that functional programs generally have many more procedure calls than do imperative programs. Note that efficiency loss here assumes the use of a uniprocessor.

1. Pure Expressions

MacLennan discusses two "worlds" within programming languages: the world of statements and the world of expressions [Ref. 24]. In the world of statements, the order in which things are evaluated is critical. A simple example of this is listed in Figure 3.2.

When assignment statements are present, it is quite possible that different sections of code within the same program will be inter-dependent. Such inter-dependencies can

```
segment A

  j:= 3;
  y:= 2;
 y:= 2j;
 y:= 3y;
 print(y)

segment B

  j:= 3;
  y:= 2;
 y:= 3y;
 y:= 2j;
 print(y)
```

Figure 3.2 Evaluation Order is Important with Statements

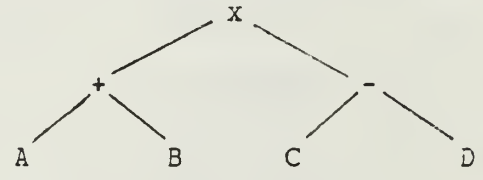
be avoided by using pure expressions. A pure expression is one which contains no assignment statements, either directly or indirectly. An example of an indirect assignment statement would be an expression which contains an assignment statement hidden in a function, such as in Figure 3.3.

Arithmetic expressions are good examples of pure expressions. In pure expressions, the operators are "memoryless", that is, the expression always has the same value within a given context. For example, in a context in which

```
3y + f(w)
where
y=3, w=2, f(x)= { y:= 4w;
                  k:= y+1;
                  x:= 2k
```

Figure 3.3 Assignment Statement Hidden in a Function

a=2, a+3 will always be 5. Moreover, the evaluation of any subexpression will have no effect on the evaluation of any other subexpression. Figure 3.4 presents a pure expression in tree form.



times[plus(A,B), minus(C,D)]

Figure 3.4 A Pure Expression

Notice that not only can the subexpressions be evaluated in any order, but (assuming the availability of more than one processor) they can be evaluated simultaneously! This is one of the big advantages that pure expressions offer parallel processors. This property of pure expressions, independence of evaluation order, is called the Church-Rosser property [Ref. 26]. It allows compilers to choose the evaluation order that will make the best use of machine resources [Ref. 24].

The evaluation of the expression starts at the leaves of the tree. The plus operator can be applied to "A" and "B" as soon as they have values. Similarly, the minus operator can be applied to "C" and "D" as soon as they have values. The times operator can be applied to the "-" and "+" nodes as soon as they have values associated with them. In more complicated expressions, we can envision values "percolating up" the tree in many different subexpressions. If the computer had many processors, then the computation of many subexpressions could be performed at the same time.

The properties of pure expressions are summarized in Figure 3.5. Many of these properties are ideally suited for programs that are to be run on a multi-processor, such as a data-flow computer. I will elaborate on some of them.

- value is independent of the evaluation order
- referential transparency
- no side effects
- inputs to an operation are obvious from the written form
- effects of an operation are obvious from the written form

Figure 3.5 Properties of Pure Expressions

As I mentioned above, independence of evaluation order is an extremely important property when it comes to concurrent processing. Recall that some imperative languages have mechanisms for evaluating different program segments in parallel, but that the burden is on the programmer to identify the critical sections. This is not at all satisfactory, because it makes the concurrent processing mechanism quite subject to programming error. Moreover, the errors which are made are not likely to be at all obvious. Rather,

they will be manifested in side effects, some of which might well escape detection even under rigorous testing of the program. In pure expressions, we are guaranteed that subexpressions can be evaluated concurrently. There are no critical sections, i.e. there is no interdependence among subexpressions! This frees the programmer from the burden of identifying the critical sections, and places the concurrency mechanism exactly where it belongs: inside the language itself.

The property of referential transparency is one which has the potential to greatly improve program efficiency. It says that a given expression (or subexpression) will always evaluate to the same value within a given context. Hence if a given expression is used several times in the same context, it need be evaluated only once! The value of the expression could be placed in a register, in a look-up table, etc. Of course, the compiler would also have the option of reevaluating the expression, if that turned out to be more efficient.

2. Pure Functions

Functions are mathematical mappings from inputs to outputs. This means that the result depends only on the inputs. If the functions are made up of pure expressions, i.e., they contain no explicit or hidden assignment statements, then the functions will retain all the properties of pure expressions. This is the basis of functional programming. Functions are applied to functions to raise the programmer to higher and higher levels of abstraction, and thus free him from as many implementation details as possible. The basis for this is pure expressions, which in turn are used to build pure functions.

3. Functional Programming

In addition to the properties of pure expressions, functional programs have some attributes which make them superior to conventional languages. Figure 3.6 lists some of these attributes.

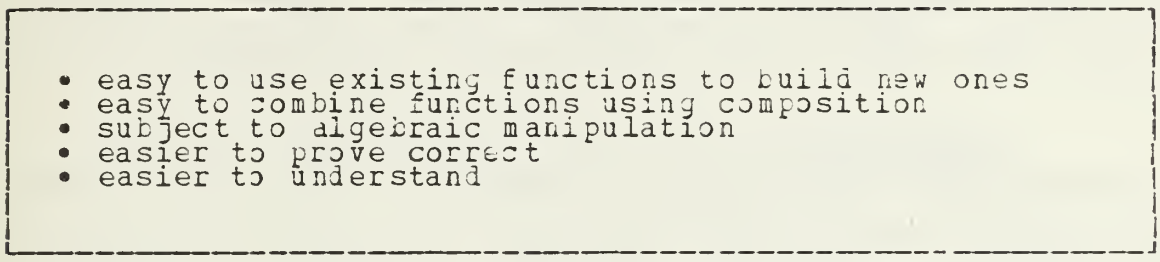


Figure 3.6 Properties of Functional Programs

The basis for most functional programming languages, including the one that I will use in my examples, is similar to that used in LISP. The functions first, rest, append, reverse, sub, null, and cons are used as an integral part of the language. If you are not familiar with these functions, I refer you to chapter two of reference [27], or to chapter nine of reference [3].

There are many notations used in functional programming. Although some people will claim that one notation is more readable than another, and others claim just the opposite, I believe that there is not really much difference among the notations. This, like many preferences, seems to be due to the system with which you have become most familiar. A similar situation exists in calculator use, where some people prefer a Hewlett-Packard calculator because it uses postfix notation, and others prefer to use Texas Instruments calculators because they use infix notation. The differences are more a matter of form than they are of substance. Similarly variations among notations in

functional programming languages really come down to syntactic sugar, and not to the expressive power of the notations. I have chosen the notation in this paper as a matter of typographical convenience.

In functional programming there is only one built in-operation: the application of a function to its argument(s) [Ref. 24]. As I pointed out earlier, plus(a,b) would apply the "plus" function to the arguments 'a' and 'b'.

Conditionals are a very natural and important part of functional programming. For example, if we want to define a function which returns the length of a list, we can do so as in Figure 3.7.

```
length L =
            if null L ---> 0
            else length(rest L) + 1
```

Figure 3.7 Conditionals in Functional Definitions

Note that the definition of length is recursive, that is, it is a function which calls itself. This is extremely common in functional programming, since to define functions explicitly (by enumeration of all input-output pairs) is not very practical.

The practice of defining functions in the fashion used in Figure 3.7 often makes the proof of correctness of functional programs much more straight forward than the proof of imperative programs. Quite often recursive functions can be proved correct by induction. Such a proof by

induction can proceed from the functions of innermost nesting, to the outermost nesting.[8]

I have mentioned that functional programming languages permit the user to work at a higher level of abstraction. For example, the map function, applies a one-argument function to every element of a list. For example, if L is a list of numbers representing angles, map sin computes the sines of the corresponding angles. Figure 3.8 is the definition of map sin.

```
map sin L =
            if null L ---> nil
            else cons[sin(first L), map sin(rest L)]
```

Figure 3.8 Mapping Across a List

Functional programs also lend themselves to algebraic manipulation. For example note in Figure 3.9 that functions often are commutative. Backus gives an excellent presentation of the algebra of functional programming languages [Ref. 4].

Functional programs seem very natural to people with a background in mathematics. The concepts of composition, reduction, transposition, identity, etc. are intuitive to these people. They can frequently learn a great deal about functional programming in a short period of time. On the other hand, the notations of functional programming languages are often such that they are not intimidating to people without a strong background in mathematics. Although most functional programming is based on the work of the

---

[8]In the length example, first the rest function would be proved correct, and then the length function would be proved.

lambda calculus, it does not adopt its intimidating symbology.

```
rest(map sin L) = map sin(rest L)
```

Figure 3.9 Algebraic Properties of FPLs

Functional programming languages are less likely to "throw away" information that the programmer has than are conventional programming languages. For example, suppose that a programmer wants to map the product reduction across a list of lists. He knows what he wants to do: he wants to use a general function which will take inputs of the form

<<2,3>, <1,4,6>, <3>, <>, <5,5>>

and produce a list like[9]

<times(2,3), times(times(1,4),6), 3, 1, times(5,5)>

which evaluates to

<6, 24, 3, 1, 25>.

Figure 3.10 shows the definition of such a function, called map_prod. In such a system, the individual product reductions of all the lists could be performed simultaneously. The programmer knows that, and indeed that can occur if he uses a functional programming language. However, if he writes his program in a conventional language, such as FORTRAN, he will be forced to write it using "Do loops". Even though he knows that the operations can be performed in parallel, that information is "hidden" from the machine. Thus operations which could be safely conducted in parallel

---

[9]Actually, each element in the list (except "<>") would call the function times once more, in the form times(x,1) where x = the element of the list.

will be performed sequentially because of language imposed limitations.

```
      prod L =
               null L ---> 1
               else times[(first L),
                          prod(rest L)]

   map_prod L =
               null L ---> nil
               else cons [prod(first L),
                          map_prod(rest L)]
```

Figure 3.10 Product Reduction Across a List of Lists

B. FUNCTIONAL PROGRAMS ON UNIPROCESSORS

Functional programming languages (and in particular the lambda calculus) were in existence long before the need for concurrent processing became apparent. As I discussed in the first chapter, programming languages should serve their (human) users. A large part of that goal can be achieved by making the language understandable to people! Henderson states that the willingness to accept less efficient but more understandable programs is a trend which will accelerate in the near future [Ref. 5]. One way to make languages more understandable is to make them simpler and more uniform. Functional programming languages, with their one built-in operation, are certainly that! Because the programmer can work at a higher level of abstraction with functional programming languages, the programs he writes can be shorter and clearer. Since software costs overwhelmingly dominate hardware costs [Ref. 28], and since the maintenance phase (including program improvement/enhancement) is the

longest phase of a program's life cycle, we can gain a great deal by using a programming language that is easy for people to understand. A carefully written functional program[10] will usually be much more readable than one written in a conventional language. That alone makes functional programming an attractive option.

Exhaustive testing of anything but a trivial program is not usually practical. Even when a program is subjected to extensive testing, "bugs" frequently are present in early versions. There are many situations, such as military applications involving nuclear weapons, when even a very low probability of program error is unacceptable. In such situations, we would like to prove the program correct before it is used. Functional programming lends itself to formal mathematical proofs. That is not to say that proofs of complicated programs are easily accomplished, even if the program is written in a functional language. However, proof of correctness is much more achieveable if the program is written in a FPL.

C. FUNCTIONAL PROGRAMMING ON A MULTIPROCESSOR

One of the tradeoffs we deal with when using functional programs on a uniprocessor is program clarity vs. program efficiency. The property of referential transparency always applies to functional programs. Therefore, even on a uniprocessor, there is a certain amount of efficiency gained. However, this will be offset many times over by the increased number of procedure calls in a functional program. So on a uniprocessor, the user gives up efficiency for

---

[10]Later in the paper I will give a comparison between a program written "mechanically" and an elegant solution. The differences are not always great. In any case, a good functional programmer should be able to easily rewrite mechanically transformed programs so that they are quite understandable.

understandability. In other words, it is acceptable for understandable programs to take longer to run.

On a multiprocessor such as a data-flow machine, efficiency must be viewed in a different light. Since the system has hundreds or even thousands of processors available for use at one time, any given program can take advantage of that only if different program parts can be run simultaneously on different processors. In functional programs, the inefficiency caused by the procedure calls is more than offset by the number of processors working on the program at any one time. Thus the independence of evaluation order plays a crucial part in the turnaround time of a program on a multiprocessor.

On a processor such as a data-flow machine, a functional program can be both more efficient and more understandable than one written in a conventional language.

D. UNDERSTANDABILITY OF FUNCTIONAL PROGRAMS

One of the principal advantages of functional programming languages is that they allow the programmer to work at a higher level of abstraction, and thus free him from many implementation details. This is accomplished through the "layering" principle. Functions are defined using previously defined functions. In this way, the primitive functions of the language, although they are implicitly included in every program, need not appear explicitly anywhere in the code.

A simple example of this layering principle is found in an exercise in Henderson's book [Ref. 5: p.280]. First we define a function which takes as argument a pair of numbers and returns as result their minimum and their maximum. This is done in Figure 3.11. Next we define a function which takes as argument three numbers and returns three results, the numbers in ascending order. This is shown in Figure 3.12.

As you can see, it is easy for the programmer to view the sort function from a higher level, such as:

1. Order the first two elements.
2. Order the second two elements.
3. Order the first two elements.

When the programmer is writing (or reviewing) the sort function, he doesn't have to be concerned with the details of how the order function works. That was done when the order function was written. Of course, this same thing can be done when working with an imperative language, but it is the very essence of functional programming.

```
order(x,y) =
   if x ≤ y then
    <x,y>
   else
    <y,x>
```

Figure 3.11 The Ordering of Two Numbers

```
sort3(a,b,c) =
 {let <a,b> = order(a,b)
  {let <b,c> = order(b,c)
   {let <a,b> = order(a,b)
    <a,b,c>}}}
```

Figure 3.12 The Sorting of Three Numbers

Even functional programs are not always easy for people to read and understand. This can be because the program is not written carefully, or because the program is terribly complex even when written in a functional language. However,

even when functional programs are complicated, they still retain the properties of pure expressions. They will be easier to prove than imperative programs, and will still lend themselves quite nicely to concurrent processing.

1. Elegant Programs

In order for a programmer to develop functional programs which are as efficient and as understandable as possible, the problem must be stripped down to its bare bones, and developed from the outset with a functional approach. This will be a time consuming process, since it will involve the same kinds of steps as does the development of an imperative program. The functional program will have the advantage that its developers will be able to work at a higher level of abstraction, but it will still take considerable effort to develop the program.

The resulting program should be one which will be extremely easy to read, and which will have all the advantages that we need in order to maximize concurrent processing. In comparison to an imperative program, it will be easier for people to understand, easier to prove correct, and will remove the burden of identifying the critical sections from the programmer. Keep in mind, however, that the development cost of this program will be of the same order of magnitude as the development cost of an imperative program to do the same job.

2. Mechanical Transformations

Let's suppose that we already have an imperative program for a certain application, and that we are satisfied with its performance from every aspect except one: speed of execution. Or suppose that we are considering buying a data-flow machine, but we don't want to "throw away" all the existing software which is written in an imperative

language. Of course, we can develop "elegant" functional programs, but that really comes down to throwing away our old software, which is something we were trying to avoid. Henderson has developed a mechanical method which takes an imperative program and transforms it into a functional one [Ref. 5: pp. 136-149]. The result is a program that has all the properties of a functional program, except that it might not be as easy to understand as the "elegant" solution. However, the development costs of mechanical transformation are nominal. I present this mechanical transformation process in the next chapter. The approach that it takes is very much like the approach that Wulf and Shaw took [Ref. 16] when they mechanically removed Go Tos from programs. Henderson's method is an excellent and inexpensive way to transform programs which will not require much maintenance, i.e. programs which have been time tested and which perform satisfactorily. They are also very readable, and hence easy to maintain, although perhaps not to the extent of the "elegant" solution.

E. ALTERNATIVES TO FUNCTIONAL PROGRAMMING

There are certainly many applications for which imperative programs will perform quite nicely. Moreover, there are some applications for which functional programs are not particularly well suited. Recall that the operators in functional programming languages are "memoryless". This means that functional programming languages are ill-suited for applications which must focus on state changes. Another argument that could be made against functional programming languages is that they are not common in industry. This is true, and is probably the very reason that COBOL and FORTRAN are still so prevalent. I do not intend to dwell on people's resistance to change, nor on the management considerations

of how to effectively implement change. I intend merely to present a reasonable argument for change, and leave the decision to the reader.

There may be cases where it would be easier and more cost effective for a firm to extend their concurrent processing capability through a language like concurrent Pascal. No doubt, such a plan would have a great deal of merit, especially when considering the costs that could be saved in programmer training. However, if such a plan were adopted, the responsibility to identify critical sections (which could lead to a whole realm of potential errors) would be placed on the shoulders of the programmer, instead of on the language itself, where it belongs. There might be errors of omission, which would result in idle CPU time, and errors of commission, which would result in potential run time errors.

Finally, I must point out that there are other "special" languages, such as VAL [Ref. 29]. VAL was developed at M.I.T. specifically for the purpose of concurrent processing. The designers have cleverly kept the assignment statement (":=") in the language, presumably so that experienced programmers would feel "at home" when they began to study it. But, the ":=" does not have the meaning of the assignment statement at all! Just as in functional programming languages, VAL uses variable free programming. McGraw refers to a single-assignment rule, which means that once an identifier is bound to a value, that binding remains in force for the entire scope of access to that identifier [Ref. 29: p.51]. This is how VAL achieves the property of evaluation order independence, and in turn why it is so well suited for concurrent processing. In my opinion, VAL is really just another another member of the functional programming language family. Its differences are slight and are mostly a matter of notation.

In the next chapter I will describe Henderson's transformation process, and extend it to handle arrays and records.

IV. FUNCTIONAL PROGRAMMING APPLICATIONS

A. FROM IMPERATIVE TO FUNCTIONAL

In this chapter I would like to suppose that we have already decided to take advantage of the processing power afforded by a multiprocessor architecture. In order to do this, we will have to employ a programming language that does not use assignment statements. For programs which are being developed for the first time, we will use the functional approach from the outset. But what about existing software that is written in an imperative language? As I pointed out in the previous chapter, we *could* develop *new* functional programs. The problem with this method is that it in no way takes advantage of the investment we made when the software was originally developed.

Henderson describes a mechanical way to transform imperative programs into functional programs [Ref. 5]. This method has the advantage that the programs which it produces contain all the properties of pure expressions, including independence of evaluation order and referential transparency. For cases in which we are satisfied with the performance of an imperative program already in our inventory, and if these programs are not subject to a great deal of change or maintenance, we could think of these as programs in an *imperative* "black box". Figure 4.1 illustrates how Henderson's method could be used to transform these programs into programs in a *functional* "black box". The resulting programs have *all* the characteristics of the original programs with respect to program correctness. Moreover, redundant assignment statements in the imperative program will be eliminated by the transformation process. Therefore

if we were satisfied with the performance of the programs in imperative form, we are guaranteed to be satisfied with their performance in functional form. The performance of the functional programs will be the same as the imperative programs, except that the functional programs can be processed on a parallel processor.

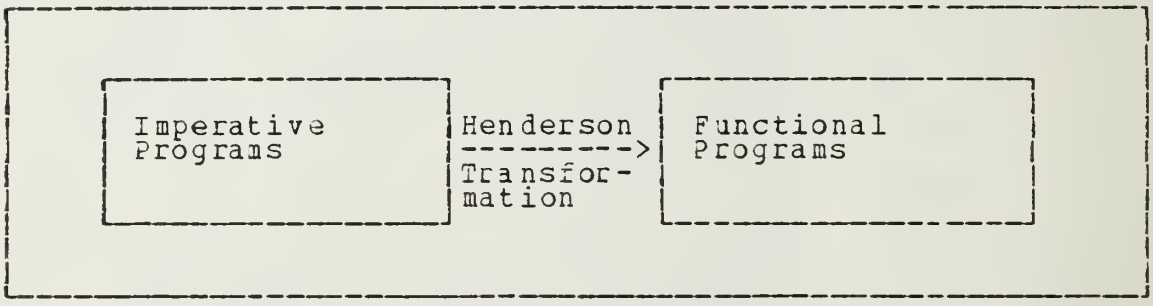


Figure 4.1 Program Transformation

In the next section, I will present the basics of Henderson's transformation process. As my basis I will use an imperative program which takes as input two positive integers, and which outputs the lesser of the two. I use this trivial program not for its application value, but only to demonstrate the transformation process. I will ignore the input/output mechanisms in the programs for now, but will comment on them in general in my conclusion. Figure 4.2 shows the imperative version of the program.

B. HENDERSON'S TRANSFORMATION PROCESS

The first step in Henderson's transformation process is to make a flow chart from the existing imperative program.[11] The flow chart for the imperative program is in Figure 4.3.

---

[11]In present day computer science circles, the use of flow charts to develop programs is not encouraged. Nevertheless, it is a very useful tool here, just as it is in Wulf and Shaw's method of eliminating Go tos.

```
procedure lesser(x,y,: integer);
var min: integer;
 begin
  If (x ≤ y) then
   min:= x;
  else
   min:= y;
  writeln(min)
 end; (*procedure min*)
```

Figure 4.2 An Imperative Program

The numbers on the edges of the flow chart correspond to the steps of the transformation process as I derive the corresponding functional program. Note that local variables in the imperative program are eliminated in the functional program.

The general procedure in the transformation process is to begin at the exit of the program (or procedure) and to work backwards to the beginning. The exit is usually represented by the identity function.[12] In this case it is the variable min.

step 1.

At (1) min is output:

{min}

When crossing a block which is an assignment statement, that which is on the right side of the assignment statement is substituted for all instances of the variable on the left side of the assignment statement which are found in the parameter list. Thus it is through parameter passing that assignment statements are handled in functional programs.

---

[12]Throughout this chapter, I will use "curly braces" ({}) to identify sections of program which have changed in any given step.

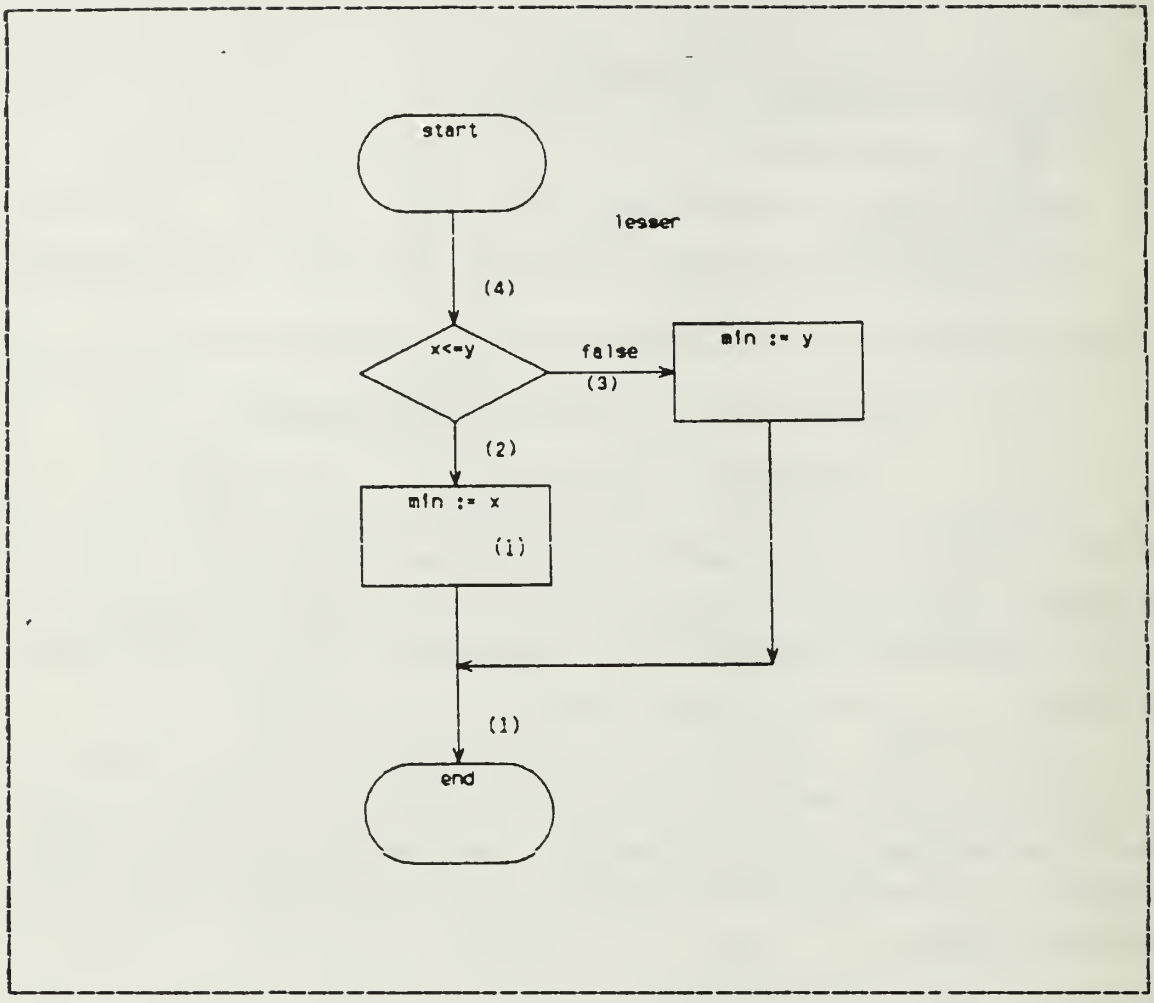


Figure 4.3 Flowchart of "lesser"

step 2.

At (2) x is substituted for all instances of min:

{x}

step 3.

At (3) y is substituted for all instances of min:

{y}

When crossing a decision block, the condition of the block is included in the code so that the program will

branch to one of two previously developed "steps". No new substitutions are made.

step 4. At (4) the program branches to either (2) or (3):

```
{If x ≤ y then}
  x
{else}
  y
```

When you have worked your way to the beginning of the flow chart, the function is defined. Figure 4.4 contains the functional definition of lesser.

```
lesser(x,y) =
              if x ≤ y then
                x
              else
                y
```

Figure 4.4 A Functional Program

C. EXTENDING THE BASIC PROCESS

In programs which have loops in them, we must have a mechanism which "cuts" the loop, or else the program would never terminate. This is done by giving each loop a function name. The flow chart is labeled with the name at the entry point. At the conclusion of the transformation process, the definitions of all the "sub-functions" will be found at the point on the chart where they are identified. For example, let's convert an imperative program which doubles a positive integer to its highest two digit number.[13] If the input

---

[13]Just as in the last example, the program I use is not

value is strictly greater than 30, it is returned as is. For example, highest(2) = 64, highest(3) = 96, highest(5) = 80, highest(150) = 150, highest(43) = 43, etc. Figure 4.5 contains the imperative program.

```
procedure highest(var x:integer);
begin
 if x > 30 then
  writeln(x)
 else
  begin
   while x < 50 do
    x:= 2x;
   writeln(x)
 end (*if x > 30...*)
end (*procedure highest*)
```

Figure 4.5 An Imperative Program with Looping

A flow chart is developed for the program (Figure 4.6). As usual, the numbers on the flow chart correspond to the steps in the transformation process.

step 1.

At (1) the output from the procedure is presented:

{x}

step 2.

At (2) the "f" loop is cut, resulting in:

{f(x)}

step 3.

At (3) 2x is substituted for all instances of x:

f({2x})

---

intended to be useful, except in how it illustrates the transformation process.

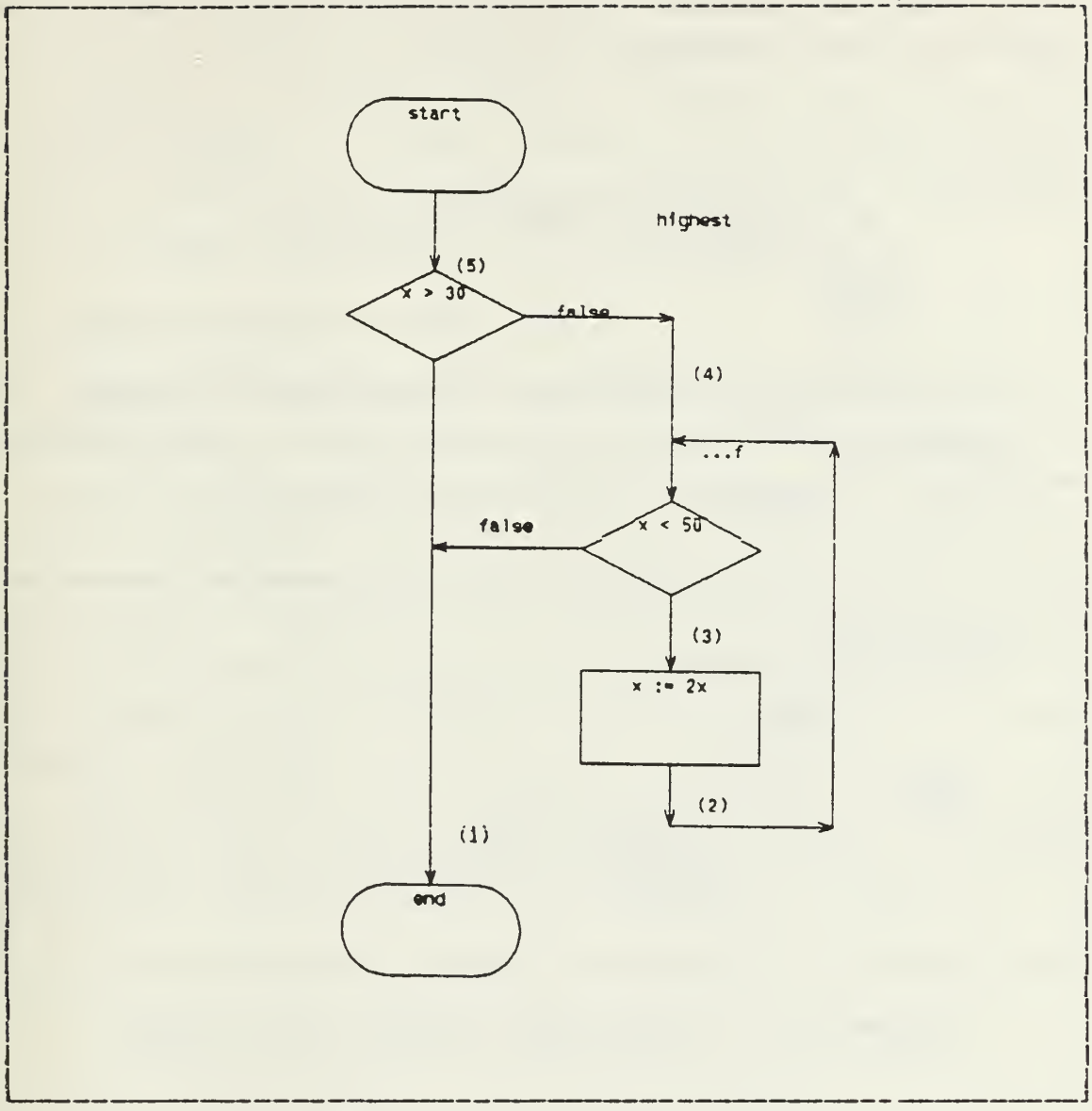


Figure 4.6 Flow Chart of "highest"

step 4.

At (4) the program branches to either (3) or (1):

```
{if x < 50 then}
   f(2x)
{else}
   x
```

Note that it is here that the function "f" is defined. We therefore will take advantage of the property of

referential transparency and only carry f(x) forward as we proceed in the transformation process.

step 5.

At (5) the program branches to (4) or (1):

```
{if x > 30 then}
   x
{else}
   f(x)
```

This gives us the functional definition of highest, if we include the definition of "f" from step 19. The complete definition of highest is contained in Figure 4.7.

```
highest(x) =
            if x > 30 then
              x
            else
              f(x)

   where
              f(x) =
                     if x < 50 then
                       f(2x)
                     else
                       x
```

Figure 4.7 A Functional Program with Looping

Note that the looping structure of the imperative program is captured in the recursive nature of the function "f".

D. TRANSFORMING "COMPLICATED" STRUCTURES

The Henderson transformation process does not take into account variables which are part of an array or record structure. The method of transformation is the same, but it is not immediately apparent how to access these variables. In the next section of the paper, I present the translation

of a Shell sort from an imperative language (Pascal) to a functional programming language. I use two functions, sub, and update, to achieve access of arrays, and to update elements therein. I handle records by dealing with them as lists of lists, and using sub to access them. The definitions of sub, and update are found in Figure 4.10.

E. TRANSFORMATION OF SHELL SORT

As a more complicated example of the Henderson transformation process, I will present the algorithm Shell sort in imperative form, and then give the step-by-step translation into a functional form. I will also present more elegant functional representations of the Shellsort, and make some comparisons. Reference [30] provides an excellent discussion of the Shell sort, although a thorough understanding of how it works is not necessary in order to follow the transformation process.

The imperative form of the Shell sort is taken from Tenenbaum and Augenstein's text on data structures [Ref. 31]. Figure 4.8 shows this program.

```
const numelts = 100;
type arraytype = array(1..numelts) of integer;
     aptr = 1..numelts
     incarray =
         record
          numinc: 1..numelts;
          incrmnts: array(1..numelts) of aptr
         end;
var x: arraytype;
    n: aptr;
procedure shell(var x:arraytype;n:aptr;inc:incarray);
var j, span: aptr
    incr, y, k: integer;
    found: boolean
begin (*procedure shell*)
 for incr := 1 to inc.numinc
  do begin
   span := inc.incrmnts(incr); (*span is the size*)
                               (*of the increment*)
   for j := span+1 to n
    do begin
     (*insert element x(j) into its proper*)
     (*   position within its subfile     *)
     y := x(j);
     k := j-span;
     found := false;
     while (k <= 1) and (not found)
      do if y < x(k)
       then begin
        x(k+span) := x(k);
        k := k-span
       end
       else found := true;
     x(k+span) := y
    end (*for...do begin*)
 end (*for...do begin*)
end (*procedure shell*)
```

Figure 4.8 Shell Sort in Imperative Form

The first step in the transformation process is to model the imperative program in a flow diagram. This is shown in Figure 4.9.

Because of the array and record structures used in the imperative algorithm, I will use the sub and update functions. In figure 4.10 these are defined, and the steps of the transformation process are labeled.

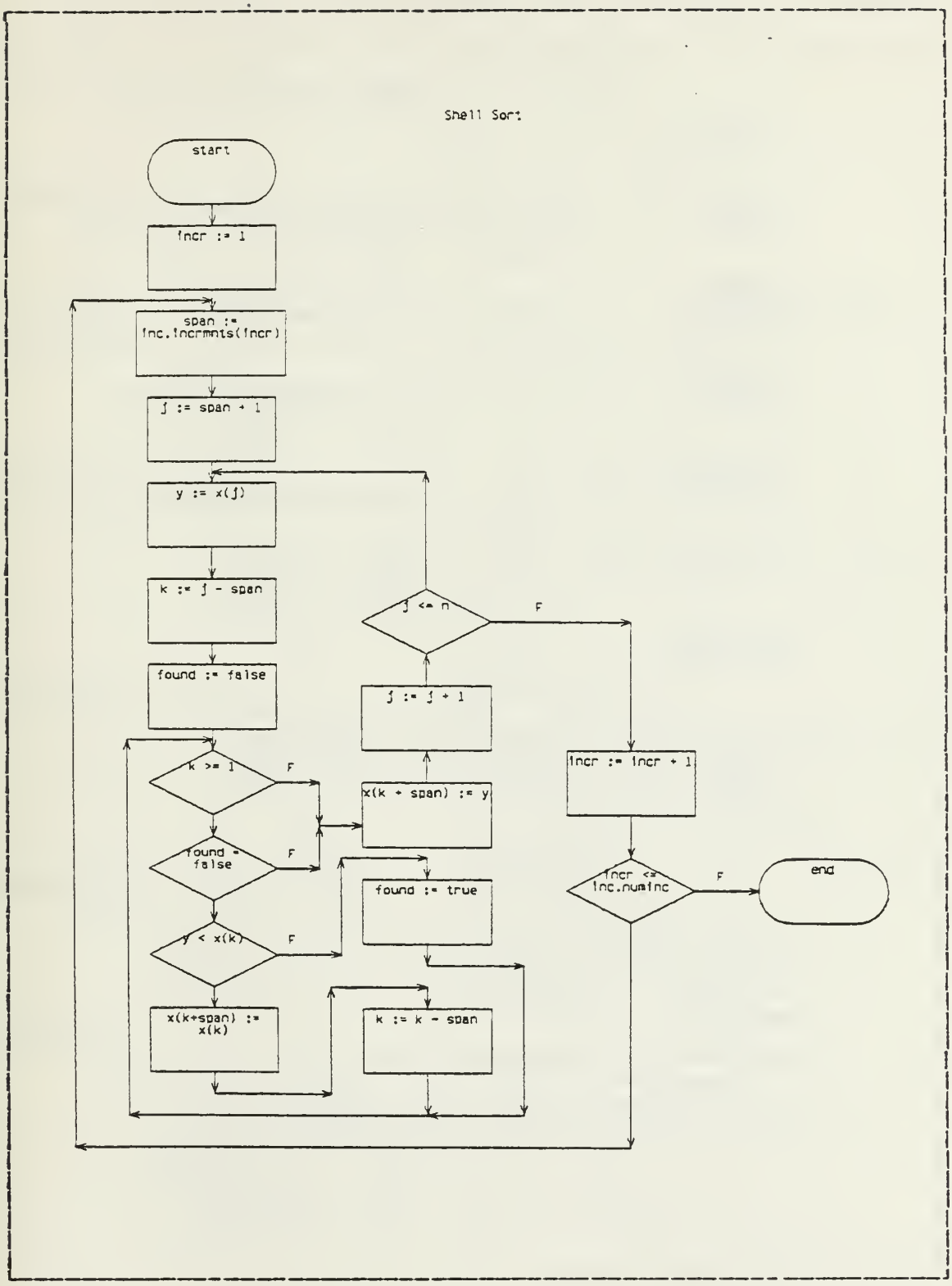


Figure 4.9 Flow Diagram of Shell Sort

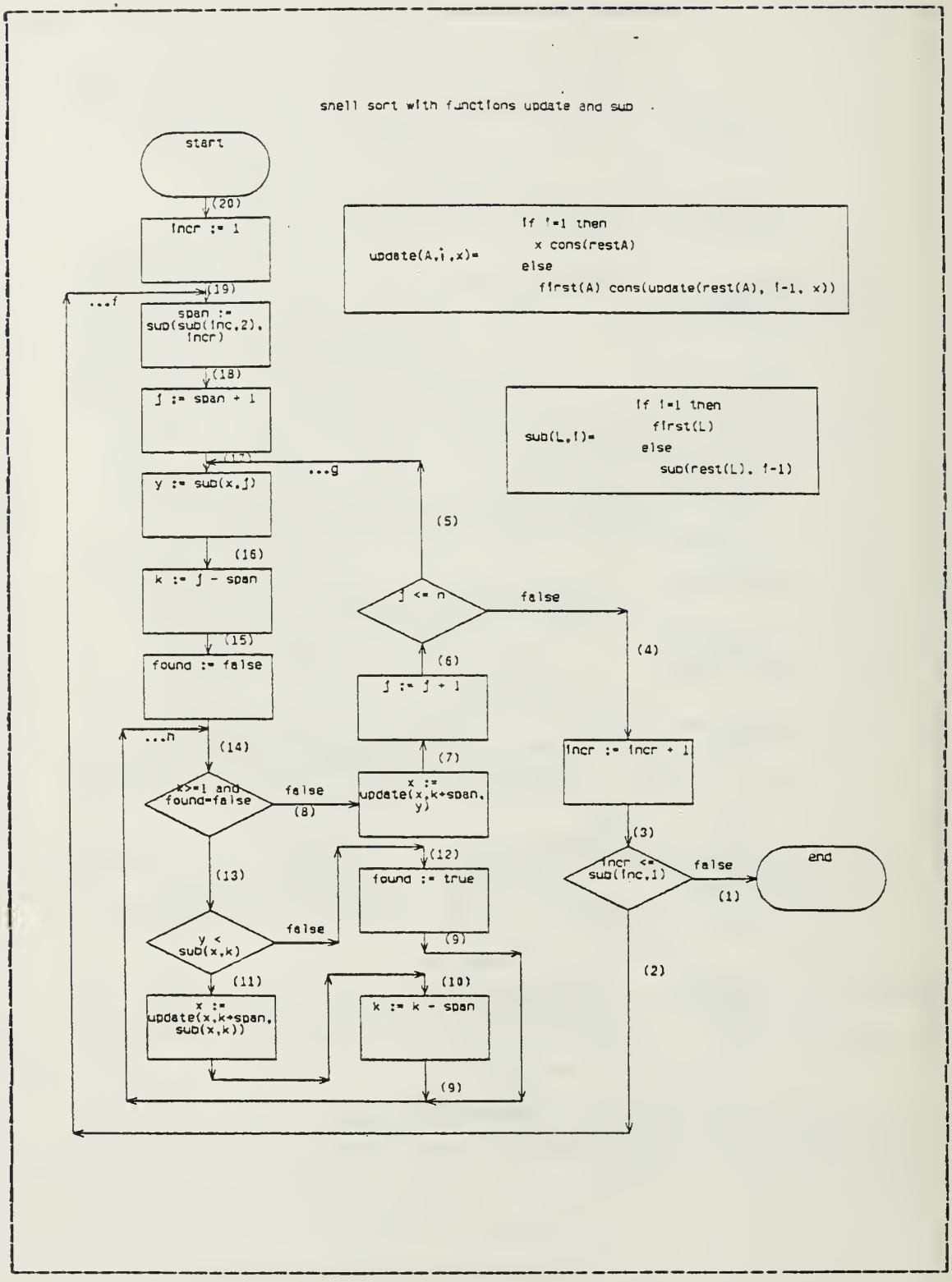


Figure 4.10 Shell Sort with "sub" and "update"

Using the same method that I have outlined in previous sections, I now present the transformation of the Shell sort into a functional language:

step 1.

At (1) the sorted array is output:

```
{x}
```

step 2.

At (2) the "f" loop is cut, resulting in:

```
{f(incr, span, j, y, k, found, x, n,
 inc)}
```

step 3.

At (3) there is a branch to either (1) or (2):

```
{if incr ≤ (sub(inc,1)) then}
  f(incr, span, j, y, k, found,
    x, n, inc)
 {else}
  x
```

step 4.

At (4) incr+1 is substituted for incr:

```
if {incr+1} ≤ (sub(inc,1)) then
 f({incr+1}, span, j, y, k, found,
   x, n, inc)
else
 x
```

step 5.

At (5) the "g" loop is cut, resulting in:

```
{g(incr, span, j, y, k, found, x,
 n, inc)}
```

step 6.

At (6) there is a branch to either (4) or (5):

```
{if j ≤ n then}
  g(incr, span, j, y, k, found, x,
    n, inc)
{else}
 if incr+1 ≤ (sub(inc,1)) then
  f(incr+1, span, j, y, k, found, x,
    n, inc)
 else
  x
```

step 7.
At (7) j+1 is substituted for j:

```
if {j+1} ≤ n then
 g(incr , span, {j+1}, y, k, found, x,
   n, inc)
else
 if incr+1 ≤ (sub(inc,1)) then
  f(incr+1 , span, {j+1}, y, k, found, x,
    n, inc)
 else
  x
```

step 8.
At (8) update(x, k+span, y) is substituted for x:

```
if j+1 ≤ n then
 g(incr , span, j+1, y, k, found,
   {(update(x, k+span, y)}, n, inc)
else
 if incr+1 ≤ (sub(inc,1)) then
  f(incr+1 , span, j+1, y, k, found,
    {update(x, k+span, y)}, n, inc)
 else
  {update(x, k+span, y)}
```

step 9.
At (9) the "h" loop is cut, resulting in:

```
{h(incr, span, j, y, k, found,
   x, n, inc)}
```

step 10.
At (10) k-span is substituted for k:

```
h(incr, span, j, y, {k-span}, found,
  x, n, inc)
```

step 11.
At (11) update(x, k+span, sub(x,k)) is substituted for x:

```
h(incr, span, j, y, k-span, found,
  {update(x, k+span, sub(x,k))},
  n, inc)
```

step 12.
At (12) true is substituted for found (from (9)):

```
h(incr, span, j, y, k, {true}, x, n, inc)
```

step 13.
At (13) there is a branch to either (11) or (12):

```
{if y < sub(x,k) then}
 h(incr, span, j, y, k-span, found,
   update(x, k+span, sub(x,k)), n,
   inc)
{else}
 h(incr, span, j, y, k, true, x, n,
   inc)
```

step 14.

At (14) there is a branch to either (13) or (8):

```
{if (k ≥ 1) and (found = false) then}
 if y < sub(x,k) then
  h(incr, span, j, y, k-span, found,
              update(x, k+span, sub(x,k)), n,
              inc)
 else
  h(incr, span, j, y, k, true, x, n, inc)
{else}
 if j+1 ≤ n then
  g(incr, span, j+1, {sub(x,j)}, k,
    found, update(x, k+span, {sub(x,j)}),
    n, inc)
 else
  if incr+1 ≤ (sub(inc,1)) then
   f(incr+1, span, j+1, y, k, found,
     update(x, k+span, y), n, inc)
  else
   update(x, k+span, y)
```

Note that it is this step that the function "h" is defined. We therefore will take advantage of the property of referential transparency and carry h(incr, span, j, y, k, found, x, n, inc) with us as we proceed in the transformation process.

step 15.

At (15) false is substituted for found:

h(incr, span, j, y, k, {false}, x, n, inc)

step 16.

At (16) j-span is substituted for k:

h(incr, span, j, y, {j-span}, false, x, n, inc)

step 17.

At (17) sub(x,j) is substituted for y:

h(incr, span, j, {sub(x,j)}, j-span, false,
  x, n, inc)

Note that it is this step that the function "g" is defined. We therefore will take advantage of the property of referential transparency and carry g(incr, span, j, y, k, found, x, n, inc) with us as we proceed in the transformation process.

step 18.

At (18) span+1 is substituted for j:

```
g(incr, span, {span+1}, y, k, found,
  x, n, inc)
```

step 19.

At (19) sub(sub(inc,2),incr) is substituted for span:

```
g(incr, {sub(sub(inc,2),incr)},
  {sub(sub(inc,2),incr)}+1, y, k,
  found, x, n, inc)
```

Note that it is this step that the function "f" is defined. We therefore will take advantage of the property of referential transparency and carry f(incr, span, j, y, k, found, x, n, inc) with us as we proceed in the transformation process.

step 20.

At (20) 1 is substituted for incr:

```
f(1, span, j, y, k, found, x, n, inc)
```

Recall that the function f is defined in step 19, the function g is defined in step 17, and the function h is defined in step 14. Figure 4.11 is the functional program for shell sort.

F. ELEGANT SOLUTIONS

An elegant solution is a program which is developed from the outset from a functional viewpoint, i.e. it does not transform an existing algorithm. The advantage of using an elegant solution is that it provides you with a custom solution to the problem, i.e. it will be designed for the specific purpose for which it is intended. That could lead to a limitation in flexibility, just as a custom wet suit is rarely useful to any diver except the one for whom it was specifically intended. But if the designer of the program

```
Shellsort(incr, span, j, y, k, found, x, n, inc) =
      f(1, span, j, y, k, found, x, n, inc)

   where f(incr, span, j, y, k, found,
           x, n, inc) =
      g(incr, sub(sub(inc,2),incr),
        sub(sub(inc,2),incr)+1, y, k,
        found, x, n, inc)

     and g(incr, span, j, y, k, found,
           x, n, inc) =
      h(incr, span, j, sub(x,j), j-span, false,
        x, n, inc)

     and h(incr, span, j, y, k, found,
           x, n, inc) =
      if (k ≥ 1) and (found = false) then
       if y < sub(x,k) then
        h(incr, span, j, y, k-span, found,
                update(x, k+span, sub(x,k)), n,
                inc)
       else
        h(incr, span, j, y, k, true, x, n, inc)
      else
       if j+1 ≤ n then
        g(incr , span, j+1, sub(x,j), k, found,
          update(x, k+span, sub(x,j)),
          n, inc)
       else
        if incr+1 ≤ (sub(inc,1)) then
         f(incr+1 , span, j+1, y, k, found,
           update(x, k+span, y), n, inc)
        else
         update(x, k+span, y)
```

Figure 4.11 The "Mechanical" Solution

keeps a broad view of the problem, the result should be easy to read and understand. It should be much easier to improve than would be an imperative program, and because of all of this, it should be easy to modify as the demands on it change.

In reference [26], Burge develops an elegant solution for the Shell sort. First he "streamlines" the algorithm, ridding it of what he identifies as minor inefficiencies.

Then he develops a functional program for that algorithm[14] [Ref. 26: p.222]. Figure 4.12 contains his solution.

```
sort 1 1 n
where rec sort a p n =
                  sort3 a p
                  sort2 a p
                  interchange a p

          and sort3 a p =
                  if a + 3p > n
                  then exit
                  else sort a (3p)
                       sort(a + p)(3p)
                       sort(a + 2p)(3p)

          and sort2 a p =
                     if a + 2p > n
                     then exit
                     else sortm a(2p)
                          sortm(a + p)(2p)
                     and sortm a p =
                               sort2 a p
                               interchange a p
and interchange a p = intch 0
          where rec intch(q) =
            if a + p + q ≤ n
            then exit
            else if A[a + q] < A[a + p + q]
                 then A[a + q] :=: A[a + p + q]
                      intch(q + 2p)
                 else intch(q + p)
```

Figure 4.12 The "Elegant" Solution

Burge uses some notation which deserve discussion. He uses the notation rec as a "flag" to indicate that the function being defined is recursive. When rec appears in a definition, both the left-hand side and the right-hand side of the definition contain the identifier. Burge calls this type of function circular [Ref. 26: p.20].

---

[14]On p.263 of reference [26], Burge states, "Most of the methods [which] have been expressed here in a functional notation can be found in the extensive literature on sorting." It seems that one should be able to infer from that statement that the sorting programs he develops are functional. This is not necessarily the case, which is a point I develop in the ensuing text.

The symbol ":=:" deserves special attention, since it appears to have all the earmarks of an assignment statement, and also appears to be at the heart of Burge's program. The ":=:" exchanges two elements of an array, i.e., A[i] :=: A[j] exchanges the ith and jth elements of array A. Thus, given an array, A, of the form:

<6,8,1,2,14>,

where a=1, p=2, and q=1,

A[a+q] :=: A[a+p+q] would result in

<6,2,1,8,14>.

This can be conceptualized in at least two ways. One way would be to use a temporary variable and to use a series of assignment statements, such as listed in Figure 4.13.

```
temp:= A[a+q];
A[a+q]:= A[a+p+q];
A[a+p+q]:= temp;,
```

Figure 4.13 Imperative Definition ":=:"

This clearly is not a functional approach and will cause us to lose the properties of referential transparency and evaluation order independence in our program.

We could also interpret the ":=:" as two successive applications of the update function.[15] This would result in code of the form:

```
update { [update (A, {a+q}, sub{A,[a+p+q]}) ],
         [a+p+q], sub[A,a+q] }
```

The effect of this code is listed in Figure 4.14.

---

[15]See Figure 4.10 for the definitions of sub and update.

1. Take as input array A.
2. Return an array A' which is the same as array A except that the (a+q)th element is the same as the (a+p+q)th element of array A.
3. Return an array A" which is the same as array A' except that the (a+p+q)th element is the same as the (a+q)th element of array A.

Figure 4.14 How the Functional ":=:" Works

This code is a little difficult to read, so now that we understand its meaning, we will make it a separate function, exchange, which "swaps" the (a+q)th and the (a+p+q)th elements of array A. Figure 4.15 contains the definition of exchange.

```
Exchange(A,a,p,q)=
    update{ [update (A, {a+q}, sub{A,[a+p+q]})],
            [a+p+q], sub A,a+q] }
```

Figure 4.15 Functional Definition of ":=:"

From a functional point of view, the meaning of ":=:" is cleared up now, but there are still some questions about Burge's "functional representation" of Shell sort. The code

then exit

appears in the program three times. This code is not semantically acceptable in functional programming! What we should be doing at these points in the program is returning the sorted array. The code for this would not be difficult

to develop,[16] but it leads us to the discovery of (from a functional programming viewpoint) another difficulty with Burge's program. The array to be sorted (presumably A), is not listed as one of the input parameters of the program. It probably is treated as a global variable, which of course leaves the code unable to stand correct on its own.

The last difficulty with Burge's elegant solution is the way he lists statements sequentially in the definition of rec sort. The program segment

```
sort3 a p
sort2 a p
interchange a p
```

would have to be changed to a functional form. This again points out the necessity to pass A to the function as an input parameter. The three functions could then be applied in the form

interchange[sort2[sort3(A,a,p),a,p],a,p].

Of course, the functions sort3, sort2, and interchange all must have an array included as an input/output parameter of their respective definitions.

All of this leads us to the unsettling and somewhat startling realization that Burge's elegant solution is recursive, easy to understand, but not functional! As I hope you will agree by my discussion, it would not be difficult to develop a purely functional program from Burge's solution, but as it stands, it is not suitable for parallel processing.

This leads me to a discussion of the dangers of using "pseudo-functional" programs.

---

[16]We could define a function exit which returns the sorted array.

G. POTENTIAL PITFALLS

When looking for an "elegant" solution, one can consult with a programmer who is expert in the art of functional programming, or consult the literature for a program which has already been developed. In the former case, it is important to ensure that the programmer knows that the program is to be used on a multiprocessor, and hence must be functionally pure. In the case of a literature search, one must be a bit more careful.

Each program which is taken "off the shelf" must be scrutinized to ensure that it doesn't have any assignment statements (explicit or hidden). It must have no sequential segments, and must have all "variables" accounted for in parameters. Many "functional" programs found in the literature will appear to be functionally pure. Nevertheless, it is important to go through the code symbol by symbol to ensure that the properties of referential transparency and independence of evaluation order independence are being preserved. Note that the code of any function that is called, but not explicitly defined, must also be scrutinized so that we can be certain that the function is based on pure expressions.

One must also be careful about using languages which are sometimes thought of as "applicative" within computer science, but which are far from "pure" in the functional sense. Perhaps the best example of this is LISP. There are versions of LISP which are suitable for concurrent processing, such as concurrent LISP [Ref. 32]. The limitations of this version of LISP are the same as the limitations of concurrent versions of other languages with imperative features. Mechanisms are created to allow the programmer to label critical sections, so that side effects will not appear during the concurrent processing. Note that

the burden is once again placed on the programmer, making the process prone to error and diminishing the chances that concurrency will be maximized. Figure 4.16 is a LISP solution to the breadth first search [Ref. 27: p.146].

```
(DEFUN BREADTH (START FINISH)
       (PROG (QUEUE EXPANSION)
             SETQ QUEUE (LIST (LIST START)))
             TRYAGAIN
             (COND ((NULL QUEUE) (RETURN NIL))
                   ((EQUAL FINISH (CAAR QUEUE))
                    (RETURN (REVERSE (CAR QUEUE)))))
             (SETQ EXPANSION (EXPAND (CAR QUEUE)))
             (SETQ QUEUE (CDR QUEUE))
             (SETQ QUEUE (APPEND QUEUE EXPANSION))
             (GO TRYAGAIN)))
```

Figure 4.16 Breadth First Search in LISP

Note that every SETQ is equivalent to an assignment statement. So although LISP has the potential to be used as a purely functional language, it is rarely used in that form. It looks functional, but is really no more functional than an ALGOL or Pascal program.

The bottom line when it comes to using functional programs to enhance concurrent processing is: be certain that the program that you are calling "functional" can be reduced to pure expressions.

V. ANALYSIS AND CONCLUSIONS

A. OVERVIEW

The assignment statement is the von Neumann bottleneck of programming languages. When describing languages which are based on pure expressions, Friedman and Wise point out that one of their most notable features is that they do not have "destructive" assignment statements, and are therefore free of side effects [Ref. 33]. This is the way in which referential transparency and independence of evaluation order are achieved. Once these attributes are present in a programming language, its expressive power (in terms of its ability to be processed in parallel) is no longer constrained. Many languages have "concurrent versions" which allow them to be processed on parallel machines. Unfortunately, these languages put the burden on the programmer to identify the critical sections. This increases the chances of programming error. Such errors would be manifested in side effects, and could go undetected until their potentially disastrous effects are felt. Functional languages do not have critical sections, and hence can take advantage of the hundreds or even thousands of processors that are becoming available because of VLSI technology.

B. MECHANICAL SOLUTIONS

When technological breakthroughs are achieved in computer science, it seems that there is a concern among those who already have large investments that their existing systems will become obsolete, and thus practically worthless overnight. Even in cases where hardware costs are reasonable

enough to be enticing, the costs of adapting existing software to the new machinery is frequently staggering, if not cost prohibitive. The mechanical means of converting imperative programs into functional ones is very attractive in this light. It is a simple process which produces programs which have all the properties of pure expressions. This means that all variable/value bindings are established as parameter/argument bindings in function linkages, and are therefore not subject to change during their lifetime. This presents an obvious opportunity for parallelism since subexpressions are independent of one another and therefore can be evaluated in any order, or simultaneously [Ref. 33].

In addition to creating code which can be processed on a parallel machine, the mechanical transformation also results in code which is easier to understand than imperative code. This is because functions are designed to be defined in terms of other functions. This leads to a "layering" effect which removes the programmer from much of the unnecessary detail of the program.

A functional program may be viewed as a set of mathematical equations which specify the solution [Ref. 34]. Even the "mechanically produced" functional program will be more suited to a proof of correctness. If an imperative program is at all complicated, it will be extremely difficulty to prove it correct. Thus this "by-product" of the transformation process is a very useful one.

C. ELEGANT SOLUTIONS

Despite the attractiveness of the mechanical transformation process, I am not recommending that it be used unless there is already a program in use which meets or exceeds the expectations being placed on it. In other cases, a new program should be designed, and a functional approach should

be used throughout its life cycle. In this way, programs can be tailored for the exact specifications for which they are intended, although the designers should take precautions to ensure that they can be extended to meet future requirements that arise during their life cycles.

When elegant solutions are used, special care must be taken to scrutinize the code to ensure that it can be reduced to pure expressions. One must be especially careful when using algorithms from the literature that are tagged "functional." There are many languages which appear to be functional which have "hidden" assignment statements. The presence of these will adulterate the program, and render it unsuitable for parallel processing as we have been discussing it.

D. EFFICIENCY

Recursive functions usually result in an exponential growth in parallelism [Ref. 35]. Functional notation naturally lends itself to recursive functions, so there will likely be a great many subexpressions which can be evaluated simultaneously. On a uniprocessor, a functional program will run much more slowly, because of all the procedure calls. Traditionally, proponents of functional programming have been willing to trade inefficiencies in their programs for greater understandability and provability. On multiprocessors, the inefficiencies caused by the procedure calls are not significant compared to the speed gained by parallel processing. The result is that, in a multiprocessing environment, functional programs are not only more understandable, but they run faster, too. Since functional languages exploit the power of multiprocessors, we can enjoy the best of both worlds!

E. A SURPRISING OUTCOME

When I started to learn the mechanical transformation process, I was convinced that the resulting code would be so complicated that it would be impossible for people to understand. Nevertheless, I reasoned that since it would still have all the properties of pure expressions, the code would be quite suitable for processing on a parallel machine. The only drawback, I supposed, would be that it would be difficult to maintain.

The first time I converted the Pascal version of Shell sort into a functional notation, I was met with code that was indeed obscure. The reason is that I failed to take advantage of the property of referential transparency when defining a function in a loop.[17] When the substitution is made, this forces the program to a higher level of abstraction, and tremendously increases the understandability of the program. Thus when a comparison is made between the mechanical solution [Figure 4.11] and the elegant solution [Figure 4.12], there isn't a great deal of difference in their readability. This makes the mechanical solution even more attractive.

F. OTHER ISSUES

The developers of VAL concluded that the most serious weakness of their language was an omission of general input/output facilities [Ref. 29: p.67]. Such a deficiency is common among functional programming languages. As is the case of VAL, the notation I have been discussing really only permits the most primitive I/O, namely, batch I/O. No I/O is actually done within the functional programs themselves.

---

[17]See steps 14, 17, and 19 in TRANSFORMATION OF SHELL SORT.

There are techniques for extending functional notation to include I/O, but they are beyond the scope of this paper.

Finally, note that I have made no reference to a garbage collection mechanism. In functional programs, structures are not overwritten. Recall that the update function creates a new array with the changed element. The "old" array is not overwritten. This process takes a great deal of memory. Thus a good garbage collector is a necessity. It must detect when structures are no longer going to be used in the program, and reclaim the memory they were using. Such mechanisms are available today, and thus the problem of making memory available for functional programs does not pose great difficulty.

G. IN A NUTSHELL

As long as the assignment statement is present in programming languages, we will not be able to take advantage of the potential processing power of the new machines that are being developed. Functional programming languages do not use assignments statements, and thus have the properties of referential transparency and independence of evaluation order. In addition, functional programs are free from side effects, lend themselves to algebraic manipulation, and are much easier to prove correct than are imperative programs.

There are many imperative programs which have added features to enhance concurrent processing, through the identification of critical sections. This places an additional burden on the programmer, and increases the likelihood for errors in the programs. The concurrency mechanism of FPLs is built into the language, and thus the need for the programmer to identify critical sections is eliminated.

Functional programming languages have long been applauded for their understandability. The ability of FPLs

to be processed in parallel has been known for some time, but only with the advent of VLSI technology, and the development of machines containing a large number of processors is the usefulness of this property really becoming apparent. Functional programming languages have the potential to completely harness the power of this new generation of machine.

Imperative programs can be mechanically transformed into functional programs. Since this can be done quickly and inexpensively, it is an attractive method for those who are considering investing in a parallel processing environment, but already have a large amount of software written in an imperative language.

LIST OF REFERENCES

1. The Wall Street Journal, August 28, 1981, p. 13.

2. Treleaven, Philip C., Isabel Gonveia Lima, "Japan's Fifth Generation Computer Systems," Computer, August, 1982.

3. MacLennan, Bruce J., Principles of Programming Languages: Design, Evaluation, and Implementation, Holt, Rinehart and Winston, 1983.

4. Backus, John, " Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra Of Programs." Communications of the ACM, August, 1978, volume 21, number 8.

5. Henderson, Peter, Functional Programming Application and Implementation, Prentice/Hall International, 1980.

6. Adams, George B. III, and Denning, Peter J. A Ten Year Plan for Concurrent Processing Research, Research Institute for Advanced Computer Science, March, 1984

7. Hoare, C.A.R., "Hints on Programming Language Design," Stamford Artificial Intelligence Laboratory Memo AIM 224, October, 1973.

8. Hoare, C.A.R., "The Emperor's Old Clothes," Communications of the ACM February, 1981, Volume 24, Number 2.

9. Winograd, Terry, "Breaking the Complexity Barrier (Again)," Proceedings of the ACM SIGPLAN-SIGIR Interface Meeting on Programming Languages-Informal Retrieval, November, 1973.

10. Leventhal, Lance A, Introduction to Microprocessors: Software, Hardware, Programming, Prentice-Hall, Inc., 1978.

11. Backus, John, "Function-Level Computing," IEEE Spectrum August, 1982.

12. Christiansen, Donald (Editor), "The Software Challenge," IEEE Spectrum August, 1982, p.22.

13. Boehm, Barry W., "Software and its Impact: A Quantitative Assessment," Datamation May, 1973.

14. Dijkstra, E.W., "Go To Statement Considered Harmful," Communications of the ACM, Vol. 11, No. 3 (March 1968) pp. 147-48.

15. Wulf, W., "A Case Against The Goto," Proceedings ACM National Conference, August, 1972.

16. Wulf, W., and Shaw, Mary, "Global Variables Considered Harmful," SIGPLAN Notices February, 1973, pp. 28-34.

17. Lerner, Eric J. (contributing editor), "Data-flow Architecture," IEEE Spectrum, April, 1984, pp. 57-62.

18. Dijkstra, E.W., "Cooperating Sequential Processes," Programming Languages, NATO Advanced Study Institute, Academic Press, 1968.

19. Bryant, Randal E., and Dennis, Jack B., "Concurrent Programming," Laboratory for Computer Science, Massachusetts Institute of Technology, October, 1968.

20. Hoare, C.A.R., "Monitors: An Operating System Structuring," Communications of the ACM, vol 17, number 10, October, 1974, pp. 549-557.

21. Hewitt, C., and Atkinson, R., "Parallelism and Synchronization in Actor Systems," Principles of Programming Languages, ACM, New York, January, 1977, pp. 267-280.

22. Booch, Grady, Software Engineering with ADA, Benjamin Cummings Publishing Company, 1983, p. 3.

23. Habermann, A. Nicko, and Perry, Dwayne E., Ada for Experienced Programmers, Addison-Wesley Publishing Company, 1983, pp.275-297.

24. MacLennan, Bruce J., Functional Programming Methodology: Theory and Practice, (tentative title), to be published by Addison-Wesley Publishing Company.

25. Deitel, Harvey M., An Introduction to Operating Systems, Addison-Wesley Publishing Company, 1983, p.82.

26. Burge, W.H., Recursive Programming Techniques, Addison-Wesley Publishing Company, 1975.

27. Winston, Patrick H., and Horn, B.K.P., LISP Addison-Wesley Publishing Company, 1981.

28. Boehm, Barry W., "Software and its Impact: A Quantitative Assessment," Datamation, May, 1973.

29. McGraw, James R., "The VAL Language: Description and Analysis," ACM Transactions on Programming Languages and Systems, Volume 4, No. 1 (January, 1982), pp.44-82.

30. Lorin, Harold, Sorting and Sort Systems, Addison-Wesley Publishing Company, 1975, pp.37-43.

31. Tenenbaum, Aaron M., and Augenstein, Mosie J., Data Structures Using Pascal, Prentice-Hall, Inc., 1981, pp.407-408.

32. Sugimoto, Shigeo, and Tabata, Koichi, and Agusa, Kiyoshi, and Ohno, Yutaka, "Concurrent Lisp on a Multi-Micro-Processor System," Proceedings of the 7th International Joint Conference on Artificial Intelligence; IJCAI 81, Van Couver, B.C., Canada, August 24-28, 1981.

33. Friedman, Daniel P. and Wise, David S., "The Impact of Applicative Programming on Multiprocessing," Proceedings of the 1976 International Conference on Parallel Processing, p.269.

34. Kennaway, J.R., and Sleep, M.R., "Parallel Implementation of Functional Languages," Proceedings of the 1982 International Conference on Parallel Processing, IEEE Computer Society Press, 1982, p.168.

35. Sleep, Ronan M., and Burton, F. Warren, "Towards A Zero Assignment Parallel Processor," IEEE Catalog Number CH1591-7/81/0000/0080, 1981.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Dudley Knox Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4.	Office of Research Administration Code 012A Naval Postgraduate School Monterey, California 93943	1
5.	Computer Technologies Curricular Office Code 37 Naval Postgraduate School Monterey, California 93943	1
6.	Bruce J. MacLennan Code 52Ml Naval Postgraduate School Monterey, California 93943	2
7.	Thomas R. McGrath 20 Shadow Lane Larchmont, New York 10538	2
8.	Captain Bradford D. Mercer, USAF Code 52Zi Naval Postgraduate School Monterey, California 93943	1